

BATTERY & BMS

CODING ON MMBED

LET'S TAKE A LOOK AT THE SYSTEM



A CELL (A BUNCH OF THESE ARE USED)

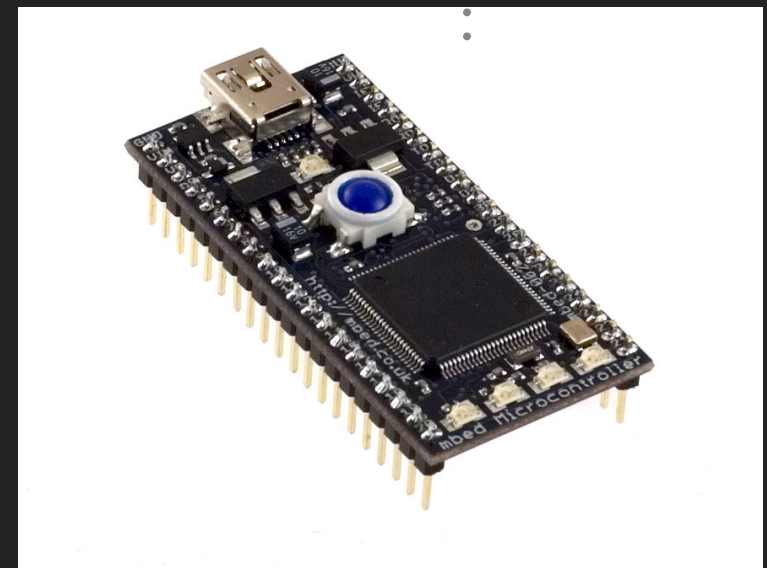
CONNECTIONS



CONNECTIONS

SOME COMPLICATED COMPONENTS (NOT REQUIRED FOR UNDERSTANDING THIS)

EMBED-
MICROCONTROLLER
(THIS IS WHERE OUR CODE GOES INTO)



WHAT CAN THE IC DO?

- Each IC can measure up to a maximum of 12 cells.
- It can do passive balancing for each of the cell.
- It can be used to measure temperature.

NOW WE ARE USING ALL THESE FUNCTIONS FOR OUR PURPOSE. THE ABOVE OVERVIEW WILL HELP US UNDERSTAND THE CONSTRUCTION OF THE CODE.

CONSTRUCTION OF THE CODE

1. Starting and setting up the SPI communication.
2. Assigning the required values to the configuration registers.
(For each functionality of the IC there exist various modes of operation. For configuring these modes we need to assign some values to these configuration registers.)
3. Now we have to start the ADC conversion of the raw voltage values and temperature values.
4. Finally we have to store the values received (temporarily) and then provide them to the microcontroller.

THE CONFIGURATION REGISTERS

The table below shows six configuration registers(CFGR0-5) with the corresponding bit wise implication.

Table 36. Configuration Register Group

REGISTER	RD/WR	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
CFGR0	RD/WR	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	REFON	SWTRD	ADCOPT
CFGR1	RD/WR	VUV[7]	VUV[6]	VUV[5]	VUV[4]	VUV[3]	VUV[2]	VUV[1]	VUV[0]
CFGR2	RD/WR	VOV[3]	VOV[2]	VOV[1]	VOV[0]	VUV[11]	VUV[10]	VUV[9]	VUV[8]
CFGR3	RD/WR	VOV[11]	VOV[10]	VOV[9]	VOV[8]	VOV[7]	VOV[6]	VOV[5]	VOV[4]
CFGR4	RD/WR	DCC8	DCC7	DCC6	DCC5	DCC4	DCC3	DCC2	DCC1
CFGR5	RD/WR	DCTO[3]	DCTO[2]	DCTO[1]	DCTO[0]	DCC12	DCC11	DCC10	DCC9

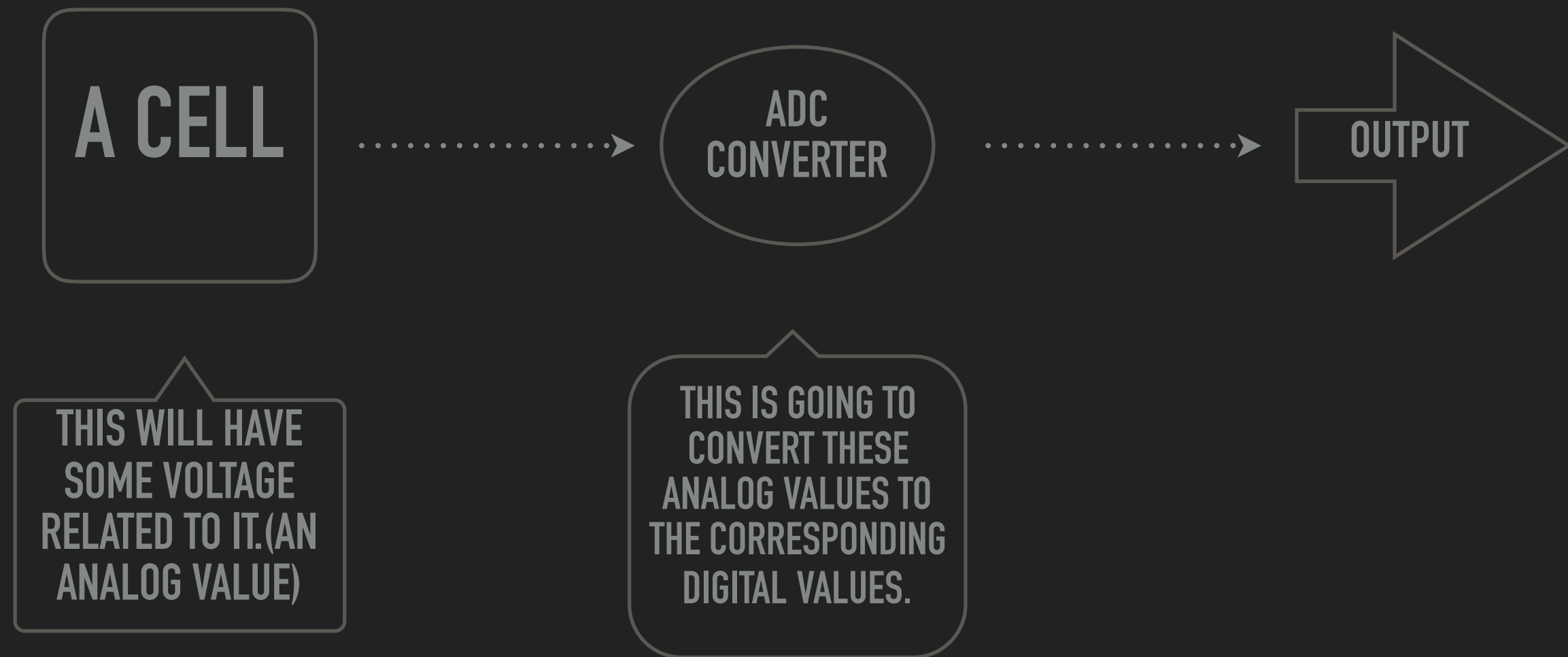
Now we will choose the value of each bit for each register step by step.

CFGR0	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Use	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	REFON	SWTRD	ADCOPT
Value used	0	0	0	0	0	1	*	0

GPIOx is 1 if xth pin used and 0 if it's not used.

REFON is 1 if 'we have to power up the reference until the watchdog timeout' and 0 if 'we have to shut down after every conversion'.

Understanding REFON in a better way



Now the thing to understand here is that the ADC converter will not be converting all the time and it needs some power (energy) to do the job. If we choose REFUP to be 1 then it will always be powered up until watchdog timer expires and if it's chosen to be 0 then it shuts down after every conversion.

THE CONFIGURATION REGISTERS

The SWTRD is read only and is used for telling the SWETEN pin status. In our case it doesn't matter if it's value is either 0 or 1.

ADCOPT is used for selecting the mode of operation of ADC. ADC can operate on 7kHz, 2kHz, 3kHz, 27kHz, 14kHz and 26kHz. Here 0 is used to select from 27, 7 or 26 kHz and 1 is used to select from 14, 3 or 2 kHz which in turn can be further selected from using the MD[1:0] Bits.

VUV and VOV refers to under voltage and over voltage comparison voltages respective. We don't use this feature. Hence all bits are assigned 0.

CFGR1	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Use	VUV[7]	VUV[6]	VUV[5]	VUV[4]	VUV[3]	VUV[2]	VUV[1]	VUV[0]
Value used	0	0	0	0	0	0	0	0

CFGR2	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Use	VOV[3]	VOV[2]	VOV[1]	VOV[0]	VUV[11]	VUV[10]	VUV[9]	VUV[8]
Value used	0	0	0	0	0	0	0	0

THE CONFIGURATION REGISTERS

CFGR3	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Use	VOV[11]	VOV[10]	VOV[9]	VOV[8]	VOV[7]	VOV[6]	VOV[5]	VOV[4]
Value used	0	0	0	0	0	0	0	0

CFGR4	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Use	DCC8	DCC7	DCC6	DCC5	DCC4	DCC3	DCC2	DCC1
Value used	0	0	0	0	0	0	0	0

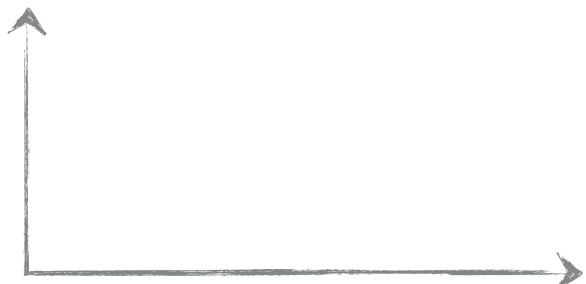
CFGR5	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Use	DCTO[3]	DCTO[2]	DCTO[1]	DCTO[0]	DCC12	DCC11	DCC10	DCC9
Value used	0	0	0	1	0	0	0	0

THE CONFIGURATION REGISTERS

DCC_i refers to whether the *i*th cell will discharge or not. 1 corresponds to it being discharged and 0 corresponding to it not being discharged. Therefore initially we let all the DCC bits to be 0.

DCTO refers to the timeout discharge value, which is matched according to the following table:

DCTO (Write)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Time (Min)	Disabled	0.5	1	2	3	4	5	10	15	20	30	40	60	75	90	120



This is the value
which we are
considering to take.

THE CODE

Now let's try to understand the various prebuilt functions provided by Linear Technology in their .cpp file.

The functions are as:

```
void LTC6804_initialize()  
{  
    quikeval_SPI_connect();  
    spi_enable(SPI_CLOCK_DIV16);  
    set_adc(MD_NORMAL, DCP_DISABLED, CELL_CH_ALL, AUX_CH_ALL);  
}
```

THIS FUNCTION IS USED FOR CONFIGURING(OR INITIALISING) ALL THE ICS ACCORDING TO THE REQUIREMENTS. IT CAN BE CLEARLY SEEN THAT IT'S CALLING OTHER VARIOUS FUNCTIONS INSIDE OF IT, THEREFORE THEIR EXACT MEANING WILL ALSO BE CLEARED LATER. ONE THING TO NOTE HERE IS THAT ALL THE VARIABLES USED INSIDE THE FUNCTIONS MUST EITHER BE GLOBAL OR PREDEFINED.

FUNCTIONS

```
void set_adc(uint8_t MD, //ADC Mode
             uint8_t DCP, //Discharge Permit
             uint8_t CH, //Cell Channels to be measured
             uint8_t CHG //GPIO Channels to be measured
            )
{
    uint8_t md_bits;

    md_bits = (MD & 0x02) >> 1;
    ADCV[0] = md_bits + 0x02;
    md_bits = (MD & 0x01) << 7;
    ADCV[1] = md_bits + 0x60 + (DCP<<4) + CH;

    md_bits = (MD & 0x02) >> 1;
    ADAX[0] = md_bits + 0x04;
    md_bits = (MD & 0x01) << 7;
    ADAX[1] = md_bits + 0x60 + CHG ;
}
```

THIS FUNCTION IS USED FOR CONFIGURING THE STATE OF THE ADC CONVERTERS. WHAT'S HAPPENING INSIDE IS THAT WITH THE VARIABLES PASSED TO THE FUNCTION, IT IS SETTING THE GLOBAL VARIABLES ADAX AND ADCV TO THE REQUIRED VALUES(THESE ARE USED WHILE SENDING SPI MESSAGES). ANOTHER THING TO UNDERSTAND IS THE MEANING ASSOCIATED WITH THE PASSED VARIABLES.

ADCV configures Cpin inputs.

ADAX configures GPIO pins.

FUNCTIONS

```
void LTC6804_adcv()  
{  
  
    uint8_t cmd[4];  
    uint16_t temp_pec;  
  
    //1  
    cmd[0] = ADCV[0];  
    cmd[1] = ADCV[1];  
  
    //2  
    temp_pec = pec15_calc(2, ADCV);  
    cmd[2] = (uint8_t)(temp_pec >> 8);  
    cmd[3] = (uint8_t)(temp_pec);  
  
    //3  
    wakeup_idle (); //This will guarantee that the LTC6804 isoSPI port is awake.  
  
    //4  
    output_low(LTC6804_CS);  
    spi_write_array(4,cmd);  
    output_high(LTC6804_CS);  
  
}
```

THIS FUNCTION STARTS THE CONVERSION OF CPIN INPUT VOLTAGES.

NOTE- TEMP_PEC IS A TEMPORARY VARIABLE MANAGING PEC ERROR(IT IS SIMILAR TO THE CHECKSUM).

LTC6804_adcv Function sequence:

1. Load adcv command into cmd array
2. Calculate adcv cmd PEC and load pec into cmd array
3. wakeup isoSPI port, this step can be removed if isoSPI status is previously guaranteed
4. send broadcast adcv command to LTC6804 stack

FUNCTIONS

```
void LTC6804_adax()  
{  
    uint8_t cmd[4];  
    uint16_t temp_pec;  
  
    cmd[0] = ADAX[0];  
    cmd[1] = ADAX[1];  
    temp_pec = pec15_calc(2, ADAX);  
    cmd[2] = (uint8_t)(temp_pec >> 8);  
    cmd[3] = (uint8_t)(temp_pec);  
  
    wakeup_idle (); //This will guarantee that the LTC6804 isoSPI port is awake.  
    output_low(LTC6804_CS);  
    spi_write_array(4,cmd);  
    output_high(LTC6804_CS);  
  
}
```

THIS FUNCTION STARTS THE CONVERSION OF GPIO PINS.

NOTE- TEMP_PEC IS A TEMPORARY VARIABLE MANAGING PEC ERROR(IT IS SIMILAR TO THE CHECKSUM).

LTC6804_adax Function sequence:

1. Load adax command into cmd array
2. Calculate adax cmd PEC and load pec into cmd array
3. wakeup isoSPI port, this step can be removed if isoSPI status is previously guaranteed
4. send broadcast adax command to LTC6804 stack

FUNCTIONS

```
uint8_t LTC6804_rdcv(uint8_t reg,                uint8_t total_ic,  
                    uint16_t cell_codes[][12])
```

THIS FUNCTION WILL SEND THE COMMAND TO READ THE CELL VOLTAGES, PARSE THE DATA AND THEN STORE THEM IN 'CELL_CODES' ARRAY.

REG: THIS VARIABLE TELL ABOUT THE GROUP OF CELL REGISTERS TO READ, WITH 0-4 CORRESPONDING TO ALL, A-D RESPECTIVELY.

**TOTAL_IC: THIS CORRESPONDS TO THE TOTAL NUMBER OF LTC6804 CONNECTED TO THE MBED.
THIS FUNCTION RETURNS 0 IF NO ERROR IS DETECTED AND -1 IF PEC ERROR IS DETECTED.**

LTC6804_rdcv Sequence

1. Switch Statement:

a. Reg = 0

- i. Read cell voltage registers A-D for every IC in the stack
- ii. Parse raw cell voltage data in cell_codes array
- iii. Check the PEC of the data read back vs the calculated PEC for each read register command

b. Reg != 0

- i. Read single cell voltage register for all ICs in stack
- ii. Parse raw cell voltage data in cell_codes array
- iii. Check the PEC of the data read back vs the calculated PEC for each read register command

2. Return pec_error flag

FUNCTIONS

```
void LTC6804_rdcv_reg(uint8_t reg,
                        uint8_t total_ic,
                        uint8_t *data
```

THIS FUNCTION IS USUALLY USED INSIDE THE PREVIOUS FUNCTION FOR STORING THE RAW DATA FROM A SINGLE CELL VOLTAGE REGISTER. NOTE- THIS DATA IS NOT PARSED.

```
int8_t LTC6804_rdaux(uint8_t reg,
                     uint8_t total_ic,
                     uint16_t aux_codes[][6]
```

THIS FUNCTION IS SIMILAR TO LTC6804_RDCV FUNCTION, THE DIFFERENCE BEING IT OPERATED ON AUXILIARY PINS.

REG: THIS VARIABLE TELLS ABOUT THE GROUP OF REGISTERS TO READ, WITH 0-2 CORRESPONDING TO ALL, A AND B RESPECTIVELY.

THE 6 ELEMENTS IN 'AUX_CODES' ARRAY CORRESPONDS TO GPIO1-5 AND VREF2 PINS' DATA.

FUNCTIONS

```
void LTC6804_rdaux_reg(uint8_t reg,
                        uint8_t total_ic,
                        uint8_t *data)
```

THIS FUNCTION IS JUST SIMILAR TO 'LTC6804_RDCV_REG' FUNCTION. THE DIFFERENCE BEING IT HANDLES AUXILIARY PINS.

```
void LTC6804_clrcell()
```

THIS COMMAND CLEARS ALL THE CELL VOLTAGE REGISTERS AND INITIALISES THEM TO 1.

```
void LTC6804_clraux()
```

THIS COMMAND CLEARS ALL THE AUXILIARY REGISTERS AND INITIALISES THEM TO 1.

FUNCTIONS

FUNCTIONS

```
void wakeup_idle()  
{  
    output_low(LTC6804_CS);  
    delayMicroseconds(10); //Guarantees the isoSPI will be in ready mode  
    output_high(LTC6804_CS);  
}
```

WAKE ISOSPI FROM IDLE STATE.

```
void wakeup_sleep()  
{  
    output_low(LTC6804_CS);  
    delay(1); // Guarantees the LTC6804 will be in standby  
    output_high(LTC6804_CS);  
}
```

WAKE LTC6804 FROM THE SLEEP STATE

FUNCTIONS

```
void spi_write_array(uint8_t len, // Option: Number of bytes to be written on the SPI port
                    uint8_t data[] //Array of bytes to be written on the SPI port
                    )
{
    for(uint8_t i = 0; i < len; i++)
    {
        spi_write((char)data[i]);
    }
}
```

THIS FUNCTION WRITES AN ARRAY OF BYTES OUT OF THE SPI PORT.

```
void spi_write_read(uint8_t tx_Data[],//array of data to be written on SPI port
                  uint8_t tx_len, //length of the tx data array
                  uint8_t *rx_data, //Input: array that will store the data read by the SPI port
                  uint8_t rx_len //Option: number of bytes to be read from the SPI port
                  )
{
    for(uint8_t i = 0; i < tx_len; i++)
    {
        spi_write(tx_Data[i]);
    }

    for(uint8_t i = 0; i < rx_len; i++)
    {
        rx_data[i] = (uint8_t)spi_read(0xFF);
    }
}
```

THIS FUNCTION READS AND WRITES A SET NUMBER OF BYTES USING THE SPI PORT.